A Scalable Nonuniform Pointer Analysis for Embedded Programs*

Arnaud Venet

NASA Ames Research Center / Kestrel Technology
Moffett Field, CA 94035, USA
arnaud@email.arc.nasa.gov

Abstract. In this paper we present a scalable pointer analysis for embedded applications that is able to distinguish between instances of recursively defined data structures and elements of arrays. The main contribution consists of an efficient yet precise algorithm that can handle multithreaded programs. We first perform an inexpensive flow-sensitive analysis of each function in the program that generates semantic equations describing the effect of the function on the memory graph. These equations bear numerical constraints that describe nonuniform points-to relationships. We then iteratively solve these equations in order to obtain an abstract storage graph that describes the shape of data structures at every point of the program for all possible thread interleavings. We bring experimental evidence that this approach is tractable and precise for real-size embedded applications.

1 Introduction

The difficulty of statically computing precise points-to information is a major obstacle to the automatic verification of real programs. Recent successes in the verification of safety-critical software [BCC+03] have been enabled in part because this class of programs makes a very restricted use of pointer manipulations and dynamic memory allocation. There are numerous pointer-intensive applications that are not safety-critical yet still require a high level of dependability like unmanned spacecraft flight control, flight data visualization or on-board network management for example. These programs commonly use arrays and linked lists to store pointers to semaphores, message queues and data packets (for interprocess communication), partitions of the memory, etc. Existing scalable pointer analyses [Ste96,FFSA98,Das00,HT01] are uniform, i.e. they do not distinguish between elements of arrays or components of recursive data structures and are therefore of little help for the verification of these programs. It is the purpose of this paper to address the problem of inferring nonuniform points-to information for embedded programs.

Few nonuniform pointer analyses have been studied in the literature. The first one has been designed by Deutsch [Deu92,Deu94] and applies to programs with

* This work was supported in part by the RTD project IST-1999-20527 DAEDALUS of the european IST FP5 programme.

explicit data type annotations. We first redesigned Deutsch's model in order to analyze languages like C in which the type information cannot be trusted to infer the shape of a data structure [Ven96,Ven99]. However both approaches rely on a costly representation of the aliasing as an equivalence relation between access paths, which makes this kind of analysis inapplicable to programs larger than a few thousand lines. We therefore designed a new semantic model [Ven02] that is both more compact and more expressive than the one based on access paths. The interest of the latter approch lies in the representation of dynamic memory allocation using numerical timestamps, which turns pointer analysis into the classical problem of computing the numerical invariants of an arithmetic program. In the case of a sequential program, various optimization techniques can be applied that break down the complexity of analyzing large arithmetic programs as described in [BCC+02,BCC+03]. In the case of multithreaded arithmetic programs however, there are no proven techniques that can cope with shared data and thread interleaving efficiently and precisely. This is a major drawback knowing that most embedded applications are multithreaded.

In this paper we present a pointer analysis based on the semantic model of [Ven02] that can infer nonuniform points-to relations for multithreaded programs. From our experience with the verification of real embedded applications we observed that collections of objects are usually manipulated in a very regular way using simple loops. Furthermore, these loops are generally controlled by local scalar variables like an array index or a pointer to the elements of a list. It is quite uncommon to find global array indices or lists that are modified across function calls. Therefore, the information flowing through this local control structure is sufficient in practice to describe exactly the layout of arrays and the shape of linked data structures. We call it the *surface structure* of a program. In the new model proposed here we first perform a flow-sensitive analysis of the surface structure that automatically discovers numerical loop invariants relating array positions and timestamps of dynamically created objects. We use these invariants to generate semantic equations that model the effect of the function on the memory. We then iteratively solve the system made of the semantic equations generated from all functions in the program. A similar approach has been applied in [WL02] for improving the precision of inclusion-based flow-insensitive pointer analyses. Our model can be seen as a natural extension to Andersen's algorithm [And94] in which variables are indexed by integers denoting array positions and timestamps, and inclusion constraints bear numerical relations between the indices of variables. We will carry on the presentation of the analysis with this analogy in mind.

The paper is organized as follows. In Sect. 2 we define the base semantic model and the surface structure of a C program. The semantics is based on timestamps to identify instances of dynamically allocated objects. Section 3 describes the abstract interpretation of the surface structure and the inference of numerical invariants. In Sect. 4 we show how to generate nonuniform inclusion constraints from the numerical relationships obtained by the analysis of the surface structure. The iterative resolution of these constraints provides us with a

global approximation of the memory graph. We describe the implementation of an analyzer for the full C language in Sect. 5 and give some experimental results from the analysis of a real device driver. We end the paper with concluding remarks and future work.

2 Base Semantic Model

In [Ven02] we have introduced a semantic model that uniquely identifies instances of dynamically allocated objects by using timestamps of the form $\langle \lambda_1, \ldots, \lambda_n \rangle$ where the λ_i are counters associated to each loop enclosing a memory allocation command. Consider for example the following piece of code:

Example 1.

```
for(i = 0; i < 10; i++)
  for(j = 0; j < 3; j++)
    a[i][j].ptr = malloc (...);
```

In that model we would consider the couple $\langle$i, j$\rangle$ as a timestamp for distinguishing between calls to the `malloc` command. In this paper we use a simplified model which folds all nested loop counters into one. In the previous example, this would result into considering the timestamp $3\mathtt{i} + \mathtt{j}$. This amounts to having one global counter λ that is incremented whenever the execution crosses a loop and is reset to 0 whenever the execution exits an outermost loop. While both models are equivalent in uniquely identifying dynamically allocated memory, the loss of information about nested loop counters may lead to imprecisions when timestamps are represented by abstract numerical lattices [Kar76,CH78,Gra91,Min01]. This is not an issue in embedded applications since almost all loops have constant iteration bounds and arrays are traversed in a regular way as in the example above. This type of loop invariants can be efficiently and exactly computed by using the reduced product [CC79] of the lattices of linear equalities [Kar76] and intervals [CC76] for example.

Because C allows the programmer to change the layout of a structured block via aggressive type casts, using symbolic data selectors like in [Ven02] for representing points-to relations is quite challenging (see [CR99] for a detailed discussion of type casting in C). In our case this would make the analysis overly complicated since we also have to manage numerical constraints that relate timestamps and positions within blocks. We choose a simple solution that consists of using a homogeneous byte-based representation of positions within memory blocks. This means that a field in a structure is identified by its byte offset from the beginning of the structure. As a consequence we must take architecture-dependent characteristics like alignment and padding into account. Fortunately, most C front-ends provide this information for free. In such a model an edge in the points-to graph has the form $\langle \mathtt{a}, o \rangle \rhd \langle \mathtt{a}', o' \rangle$ where a, a′ are addresses of blocks in memory and o, o' are byte offsets within these blocks.

Our purpose is to abstract a C program into a system of points-to equations expressed by inclusion constraints similarly to Andersen's analysis [And94]. Since

$$
\begin{array}{lll}
\mathit{Stmt} ::= \mathtt{n} = c \quad (c \in \mathbb{N}) & | \ \mathtt{p} = *\mathtt{q} \\
\quad | \ \mathtt{n} = \mathtt{m} + \mathtt{o} & | \ *\mathtt{p} = \mathtt{q} \\
\quad | \ \mathtt{n} = \mathtt{m} * \mathtt{o} & | \ \mathtt{p} = \mathtt{malloc}() \\
\quad | \ \mathtt{p} = \&\mathtt{x} & | \ \mathtt{while}\ (\mathtt{m} < \mathtt{n})\ \mathtt{do}\ s_1; \cdots; s_n\ \mathtt{end} \\
\quad | \ \mathtt{p} = \mathtt{q} + \mathtt{n}
\end{array}
$$

Fig. 1. Syntax of the core pointer language

we want to express nonuniform aliasing relationships, we need to assign position and timestamp indices to semantic variables and relate them by using numerical constraints. For example, we would like to generate an inclusion constraint for the piece of code of Example 1 that looks like:

$$*(\&\mathtt{a} + (i \times s + o_{\mathtt{ptr}})) \supseteq \mathbf{malloc}_t \quad \text{where } i = t \ \wedge \ t \in [0, 29]$$

where s is the size of the structure contained in the two-dimensional array, $o_{\mathtt{ptr}}$ is the offset of the field ptr in that structure and t is the timestamp of the memory allocation statement. In order to infer this kind of constraint we must first perform a flow-sensitive analysis over a relational numerical lattice [Kar76,CH78,Gra91,Min01] that computes invariants relating loop counters, array indices and timestamps. The main difference from [Ven02] comes from the fact that we generate inclusion constraints without any prior knowledge of the layout of objects in the heap. In this case it is not obvious what to do with the following piece of code:

Example 2.

```
for(i = 0; i < 10; i++) {
  p = p->next;
}
```

The rest of this section will be devoted to defining a concrete semantic model that will allow us to handle this situation simply and precisely.

We base our semantic specification on a small language that captures the core pointer arithmetic of C at the function level. The treatment of interprocedural mechanisms is postponed until Sect. 4 where we will detail the generation of inclusion constraints. We call *surface variable* a variable which has a scalar type, either integer or pointer, and which does not have its address taken. The syntax of the language is defined in Fig. 1, where we denote by $\mathtt{p}, \mathtt{q}, \mathtt{r}$ pointer-valued surface variables, by $\mathtt{m}, \mathtt{n}, \mathtt{o}$ integer-valued surface variables, and by $\mathtt{x}, \mathtt{y}, \mathtt{z}$ all other variables. We assume that the variable on the left handside of an assignment operation does not appear on the right handside. This will facilitate the design of the numerical abstract interpretation in Sect. 3. It is always possible to rewrite the program in order to satisfy this assumption. Note that in order to keep the presentation simple, we focus on fundamental arithmetic operations and loops. All other constructs can be analyzed along the same lines. We use

this language to model the computations that occur locally within the body of a C function, excluding calls to other functions. A program P in this language is just a sequence of statements describing the pointer manipulations performed by a function. We provide P with a small-step operational semantics given by a transition system $(\Sigma, \rightarrow)$ defined as follows.

We first need some notations. We assume that each statement of P is assigned a unique label ℓ. If ℓ is the label of a statement, we denote by $\mathbf{next}(\ell)$ the label of the next statement of P to be executed in the natural execution order. If ℓ is the label of a loop we denote by $\mathbf{top}(\ell)$ the predicate that is true iff the statement at ℓ is an outermost loop. A state of Σ is a tuple $\langle \lambda, M, \varrho, \ell \rangle$ where λ is an integer denoting the global loop counter used for timestamping, M is a memory graph, ϱ is an environment and ℓ is the label of the next statement to execute. A memory graph is a collection of points-to edges $\langle \mathrm{a}, o \rangle \rhd \langle \mathrm{a}', o' \rangle$ where a, a' are addresses and o, o' are integers representing byte offsets. An address is either the location of a global variable &x or a dynamically allocated block $\mathbf{blk}_\ell\langle t \rangle$, where ℓ is the location of the allocation statement and t is a timestamp. We use a special address **null** to represent the `NULL` pointer value in C. The mapping defined by a memory graph is functional, i.e. there is at most one outcoming edge for each memory location $\langle \mathrm{a}, o \rangle$. We denote by $M\langle \mathrm{a}, o \rangle$ the target location of the edge originating from the location $\langle \mathrm{a}, o \rangle$ if it exists or $\langle \mathbf{null}, 0 \rangle$ otherwise. We denote by $M[\langle \mathrm{a}, o \rangle \rhd \langle \mathrm{a}', o' \rangle]$ the memory graph M which has been updated with the edge $\langle \mathrm{a}, o \rangle \rhd \langle \mathrm{a}', o' \rangle$.

We split down each pointer variable p into two variables p_a and p_o that respectively denote the address of the block and the offset within this block to which p points. An environment ϱ maps variables n, p_o to integers and variables p_a to addresses. We denote by $\varrho[u \leftarrow v]$ the environment ϱ in which the variable u has been assigned the value v. Finally, we denote by Ω a special element of Σ representing the error state. The transition relation $\rightarrow$ of the operational semantics is then defined in Fig. 2. An initial state in this operational semantics assigns arbitrary integer values to surface integer variables and the null memory location to surface pointer variables. This amounts to considering integer variables as uninitialized and pointers initialized to `NULL`. For consistency the initial value of λ should be 0. In our framework an initial state describes the memory configuration at the entry of the C function that is modeled by the program P.

The transition rule for loop exits requires some explanations. The global loop counter λ is incremented at the end of each loop iteration and decremented whenever the execution steps out of a nested loop. Whether the global loop counter is decremented or left unchanged at loop exit has no effect on the uniqueness of timestamps. However decrementation is required in order to preserve linear relationships between λ and byte offsets during the traversal of multidimensional arrays. Consider the two nested loops of Example 1. We keep the previous notations and we denote by O the byte offset within a on the lefthand side of the assignment. Then, the relation between O and the loop counters is given by $O = 3 \times s \times \mathrm{i} + s \times \mathrm{j} + o_{\mathrm{ptr}}$. If we use the decrementation rule at loop exit, the global loop counter value is given by $\lambda = 3 \times \mathrm{i} + \mathrm{j}$, hence $O = s \times \lambda + o_{\mathrm{ptr}}$.

$$\langle \lambda, M, \varrho, \ell : \mathtt{n} = c \rangle \rightharpoonup \langle \lambda, M, \varrho[\mathtt{n} \leftarrow c], \mathbf{next}(\ell) \rangle$$
$$\langle \lambda, M, \varrho, \ell : \mathtt{n} = \mathtt{m} + \mathtt{o} \rangle \rightharpoonup \langle \lambda, M, \varrho[\mathtt{n} \leftarrow \varrho(\mathtt{m}) + \varrho(\mathtt{o})], \mathbf{next}(\ell) \rangle$$
$$\langle \lambda, M, \varrho, \ell : \mathtt{n} = \mathtt{m} * \mathtt{o} \rangle \rightharpoonup \langle \lambda, M, \varrho[\mathtt{n} \leftarrow \varrho(\mathtt{m}) \times \varrho(\mathtt{o})], \mathbf{next}(\ell) \rangle$$
$$\langle \lambda, M, \varrho, \ell : \mathtt{p} = \&\mathtt{x} \rangle \rightharpoonup \langle \lambda, M, \varrho[\mathtt{p}_o \leftarrow 0, \mathtt{p}_a \leftarrow \&\mathtt{x}], \mathbf{next}(\ell) \rangle$$
$$\langle \lambda, M, \varrho, \ell : \mathtt{p} = \mathtt{q} + \mathtt{n} \rangle \rightharpoonup \langle \lambda, M, \varrho[\mathtt{p}_o \leftarrow \mathtt{q}_o + \varrho(\mathtt{n}), \mathtt{p}_a \leftarrow \varrho(\mathtt{q}_a)], \mathbf{next}(\ell) \rangle$$
$$\langle \lambda, M, \varrho, \ell : \mathtt{p} = *\mathtt{q} \rangle \rightharpoonup \begin{cases} \Omega \text{ if } \varrho(\mathtt{q}_a) = \mathbf{null} \\ \langle \lambda, M, \varrho[(\mathtt{p}_a, \mathtt{p}_o) \leftarrow M\,\langle \varrho(\mathtt{q}_a), \varrho(\mathtt{q}_o) \rangle], \mathbf{next}(\ell) \rangle & \text{otherwise} \end{cases}$$
$$\langle \lambda, M, \varrho, \ell : \mathtt{p} = \mathtt{malloc}() \rangle \rightharpoonup \langle \lambda, M, \varrho[\mathtt{p}_a \leftarrow \mathbf{blk}_\ell\langle\lambda\rangle, \mathtt{p}_o \leftarrow 0], \mathbf{next}(\ell) \rangle$$
$$\langle \lambda, M, \varrho, \ell : *\mathtt{p} = \mathtt{q} \rangle \rightharpoonup \begin{cases} \Omega \text{ if } \varrho(\mathtt{p}_a) = \mathbf{null} \\ \langle \lambda, M[\langle \varrho(\mathtt{p}_a), \varrho(\mathtt{p}_o) \rangle \rhd \langle \varrho(\mathtt{q}_a), \varrho(\mathtt{q}_o) \rangle], \varrho, \mathbf{next}(\ell) \rangle & \text{otherwise} \end{cases}$$
$$\langle \lambda, M, \varrho, \ell : \mathtt{while}\ (\mathtt{m} < \mathtt{n})\ \mathtt{do}\ \ell' : s_1; \cdots\ \mathtt{end} \rangle \rightharpoonup \langle \lambda, M, \varrho, \ell' \rangle \quad \text{if } \varrho(\mathtt{m}) < \varrho(\mathtt{n})$$
$$\langle \lambda, M, \varrho, \ell : \mathtt{while}\ (\mathtt{m} < \mathtt{n})\ \mathtt{do}\ \ldots\ \mathtt{end} \rangle \rightharpoonup \begin{cases} \langle 0, M, \varrho, \mathbf{next}(\ell) \rangle \text{ if } \varrho(\mathtt{m}) \geq \varrho(\mathtt{n}) \text{ and } \mathbf{top}(\ell) \\ \langle \lambda - 1, M, \varrho, \mathbf{next}(\ell) \rangle \text{ otherwise} \end{cases}$$
$$\langle \lambda, M, \varrho, \ell : \mathtt{end} \rangle \rightharpoonup \langle \lambda + 1, M, \varrho, \ell' : \mathtt{while}\ (\ldots)\ \mathtt{do}\ \ldots\ \ell : \mathtt{end} \rangle$$

Fig. 2. Operational semantics of the core pointer language

Without this rule λ would be equal to $4 \times \mathtt{i} + \mathtt{j}$ and the relationship between the global loop counter and O would be lost, thereby preventing the inference of a nonuniform points-to relation.

This operational semantics is similar to the one described in [Ven02] with a simplified timestamping. We need to instrument the semantics by adding an intermediate layer between the environment and the memory that keeps track of all memory accesses. Whenever a location is retrieved from the memory, we use a timestamp to tag it with a unique name that we call an *anchor*, and we keep the binding between this anchor and the actual memory location in a separate structure A called the *anchorage*. The local environment ϱ now maps the address component of a surface variable $\mathtt{p}_a$ either to an address that explicitly appears in the body of a C function or to an anchor. We call this refined semantics the *surface semantics*. More formally, the surface semantics $(\Sigma_s, \rightharpoonup_s)$ of a program P is defined as follows. A extended state of Σ_s is a tuple $\langle \lambda, A, M, \varrho, \ell \rangle$ where $\langle \lambda, M, \varrho, \ell \rangle \in \Sigma$ and A is an anchorage. An anchor $\mathbf{ref}_\ell\langle t \rangle$ denotes the value returned by the execution of a memory read command $\ell : \mathtt{p} = *\mathtt{q}$ at program point ℓ on time t. The anchorage maps an anchor $\mathbf{ref}_\ell\langle t \rangle$ to an actual memory location $\langle \mathtt{a}, o \rangle$. If $\langle a, o \rangle$ is a location stored in the environment ρ, a may either be an address or an anchor. We define the resolution function $\mathbf{get}_A$ which maps $\langle a, o \rangle$ to the corresponding memory location as follows:

$$\mathbf{get}_A\langle a, o \rangle = \begin{cases} \langle \mathbf{null}, 0 \rangle & \text{if } a \text{ is an anchor and } A(a) = \langle \mathbf{null}, 0 \rangle \\ \langle \mathtt{a}, o + o' \rangle & \text{if } a \text{ is an anchor and } A(a) = \langle \mathtt{a}, o' \rangle \\ \langle \mathtt{a}, o \rangle & \text{if } a \text{ is an address } \mathtt{a} \end{cases}$$

If p is a surface pointer and ϱ is an environment, we denote by $\mathbf{get}_{A,\varrho}(\mathtt{p})$ the memory location $\mathbf{get}_A\langle \varrho(\mathtt{p}_a), \varrho(\mathtt{p}_o) \rangle$. The transition relation $\rightharpoonup_s$ of the surface semantics is then defined in Fig. 3. The error state in this semantics is also

$$\langle \lambda, A, M, \varrho, \ell : \mathrm{p} = *\mathrm{q} \rangle \rightarrow_s \begin{cases} \Omega & \text{if } \mathbf{get}_{A,\varrho}(\mathrm{q}) = \langle \mathbf{null}, o \rangle \\ \left\langle \begin{array}{l} \lambda, A[\mathbf{ref}_\ell\langle\lambda\rangle \leftarrow M(\mathbf{get}_{A,\varrho}(\mathrm{q}))], \\ M, \varrho[\mathrm{p}_a \leftarrow \mathbf{ref}_\ell\langle\lambda\rangle, \mathrm{p}_o \leftarrow 0], \mathbf{next}(\ell) \end{array} \right\rangle & \text{otherwise} \end{cases}$$

$$\langle \lambda, A, M, \varrho, \ell : *\mathrm{p} = \mathrm{q} \rangle \rightarrow_s \begin{cases} \Omega & \text{if } \mathbf{get}_{A,\varrho}(\mathrm{p}) = \langle \mathbf{null}, o \rangle \\ \left\langle \begin{array}{l} \lambda, A, M[\mathbf{get}_{A,\varrho}(\mathrm{p}) \rhd \mathbf{get}_{A,\varrho}(\mathrm{q})], \\ \varrho, \mathbf{next}(\ell) \end{array} \right\rangle & \text{otherwise} \end{cases}$$

For all other statements:

$$\frac{\langle \lambda, M, \varrho, \ell \rangle \rightarrow \langle \lambda', M', \varrho', \ell' \rangle}{\langle \lambda, A, M, \varrho, \ell \rangle \rightarrow_s \langle \lambda', A, M', \varrho', \ell' \rangle}$$

Fig. 3. Surface semantics of the core pointer language

denoted by Ω. An initial state in the surface semantics is simply an initial state in the base semantics with an empty anchorage. We denote by I the set of all initial states.

We are interested in the *collecting semantics* [Cou81] of a program P, that is the set $C = \{i \xrightarrow{*}_s s \mid i \in I\}$ of all states reachable from any initial state I. We define the *surface structure* $\mathcal{S}$ of P as follows:

$$\mathcal{S} = \{\langle \lambda, \varrho, \ell \rangle \mid \exists M\ \exists A : \langle \lambda, A, M, \varrho, \ell \rangle \in C\}$$

An element $\langle \lambda, \varrho, \ell \rangle$ is called a *surface configuration*. The program P models the pointer manipulations performed by a single C function. Our purpose is to compute a global approximation of the memory for a whole C program by first performing an abstract interpretation of the surface structure of each function in the program. The design of this abstract interpretation is straightforward because the surface structure is independent from the data stored in the heap and does not interfere with other threads. We will then generate inclusion constraints from the results of the analysis of the surface structure that will provide us with a global approximation of the memory and the anchorage structure as well.

3 Abstract Interpretation of the Surface Structure

We describe the analysis of the surface structure within the framework of Abstract Interpretation [CC77,CC79,Cou81,CC92]. We define an abstract environment by a pair $\langle \nu^\sharp, \pi^\sharp \rangle$ as follows:

- The component $\nu^\sharp$ is an abstract numerical relation belonging to a given numerical lattice $\mathcal{V}^\sharp$ [Kar76,CH78,Gra91,Min01] that we leave as a parameter of our analysis. The abstract relation $\nu^\sharp$ is a collection of numerical constraints between all integer valued variables n, p_o of the program and a special variable Λ denoting the value of the global loop counter.
- The component $\pi^\sharp$ maps every variable p_o to a set of abstract addresses.

An abstract address is either the address of a global variable &x, a dynamically allocated block $\mathbf{blk}^\sharp_\ell\langle\mu^\sharp\rangle$ or an anchor $\mathbf{ref}^\sharp_\ell\langle\mu^\sharp\rangle$, where $\mu^\sharp$ is a abstract numerical

$$
\begin{array}{l}
[\![\mathtt{n} = c]\!]^\sharp \langle \nu^\sharp, \pi^\sharp \rangle = \langle (\nu^\sharp \ominus \{\mathtt{n}\}) \oplus \{\mathtt{n} = c\}, \pi^\sharp \rangle \\
[\![\mathtt{n} = \mathtt{m} + \mathtt{o}]\!]^\sharp \langle \nu^\sharp, \pi^\sharp \rangle = \langle (\nu^\sharp \ominus \{\mathtt{n}\}) \oplus \{\mathtt{n} = \mathtt{m} + \mathtt{o}\}, \pi^\sharp \rangle \\
[\![\mathtt{n} = \mathtt{m} * \mathtt{o}]\!]^\sharp \langle \nu^\sharp, \pi^\sharp \rangle = \langle (\nu^\sharp \ominus \{\mathtt{n}\}) \oplus \{\mathtt{n} = \mathtt{m} \times \mathtt{o}\}, \pi^\sharp \rangle \\
[\![\mathtt{p} = \&\mathtt{x}]\!]^\sharp \langle \nu^\sharp, \pi^\sharp \rangle = \langle (\nu^\sharp \ominus \{\mathtt{p}_o\}) \oplus \{\mathtt{p}_o = 0\}, \pi^\sharp[\mathtt{p}_a \leftarrow \{\&\mathtt{x}\}] \rangle \\
[\![\mathtt{p} = \mathtt{q} + \mathtt{n}]\!]^\sharp \langle \nu^\sharp, \pi^\sharp \rangle = \langle (\nu^\sharp \ominus \{\mathtt{p}_o\}) \oplus \{\mathtt{p}_o = \mathtt{q}_o + \mathtt{n}\}, \pi^\sharp[\mathtt{p}_a \leftarrow \pi^\sharp(\mathtt{q}_a)] \rangle \\
[\![\ell : \mathtt{p} = *\mathtt{q}]\!]^\sharp \langle \nu^\sharp, \pi^\sharp \rangle = \left\langle \begin{array}{l} (\nu^\sharp \ominus \{\mathtt{p}_o\}) \oplus \{\mathtt{p}_o = 0\}, \\ \pi^\sharp[\mathtt{p}_a \leftarrow \{\mathbf{ref}^\sharp_\ell \langle \lfloor \nu^\sharp \oplus \{\tau = \lambda\} \rfloor_{\tau,\lambda} \rangle\}] \end{array} \right\rangle \\
[\![\ell : \mathtt{p} = \mathtt{malloc}()]\!]^\sharp \langle \nu^\sharp, \pi^\sharp \rangle = \left\langle \begin{array}{l} (\nu^\sharp \ominus \{\mathtt{p}_o\}) \oplus \{\mathtt{p}_o = 0\}, \\ \pi^\sharp[\mathtt{p}_a \leftarrow \{\mathbf{blk}^\sharp_\ell \langle \lfloor \nu^\sharp \oplus \{\tau = \lambda\} \rfloor_{\tau,\lambda} \rangle\}] \end{array} \right\rangle \\
[\![*\mathtt{p} = \mathtt{q}]\!]^\sharp \langle \nu^\sharp, \pi^\sharp \rangle = \langle \nu^\sharp, \pi^\sharp \rangle
\end{array}
$$

Fig. 4. Abstract surface semantics of atomic statements

relation between the loop counter variable Λ and a special timestamp variable denoted by τ. We assume that for each set of abstract addresses, there is at most one abstract address $\mathbf{blk}^\sharp_\ell\langle\mu^\sharp\rangle$ or $\mathbf{ref}^\sharp_\ell\langle\mu^\sharp\rangle$ per program location ℓ. Therefore, the set $E^\sharp$ of all abstract environments is isomorphic to the product $\prod_{i\in I} \mathcal{V}^\sharp$ of the numerical lattice over a fixed family I. We provide $E^\sharp$ with the structure of a lattice by lifting all operations of $\mathcal{V}^\sharp$ to $E^\sharp$ pointwise.

The denotation $\gamma_{\mathcal{V}^\sharp}(\nu^\sharp)$ of an abstract numerical relation is a set of variable assignments ε that satisfy the numerical constraints expressed by $\nu^\sharp$. If $x_1, \ldots, x_n$ are numerical variables and $v_1, \ldots, v_n$ are integer values, we denote by $\nu^\sharp\langle x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\rangle$ the predicate that is true iff there is an assignment $\varepsilon \in \gamma_{\mathcal{V}^\sharp}(\nu^\sharp)$ such that $\varepsilon(x_i) = v_i$ for all $1 \leq i \leq n$. The denotation $\gamma_{E^\sharp}\langle\nu^\sharp, \pi^\sharp\rangle$ of an abstract environment is the set of all pairs $\langle\lambda, \varrho\rangle$ where $\lambda \in \mathbb{N}$ and ϱ is an environment of the surface semantics, such that:

- $\nu^\sharp\langle \mathtt{n} \mapsto \varrho(\mathtt{n}), \ldots, \mathtt{p}_o \mapsto \varrho(\mathtt{p}_o), \ldots, \Lambda \mapsto \lambda\rangle$ for all variables $\mathtt{n}, \ldots, \mathtt{p}, \ldots$ of the program
- $\varrho(\mathtt{p}_a) = \&\mathtt{x} \Rightarrow \&\mathtt{x} \in \pi^\sharp(\mathtt{p}_a)$
- $\varrho(\mathtt{p}_a) = \mathbf{blk}_\ell\langle t\rangle \Rightarrow \mathbf{blk}^\sharp_\ell\langle\mu^\sharp\rangle \in \pi^\sharp(\mathtt{p}_a) \wedge \mu^\sharp\langle\tau \mapsto t, \Lambda \mapsto \lambda\rangle$
- $\varrho(\mathtt{p}_a) = \mathbf{ref}_\ell\langle t\rangle \Rightarrow \mathbf{ref}^\sharp_\ell\langle\mu^\sharp\rangle \in \pi^\sharp(\mathtt{p}_a) \wedge \mu^\sharp\langle\tau \mapsto t, \Lambda \mapsto \lambda\rangle$

An abstract surface configuration of the program is a family $\langle\nu^\sharp_\ell, \pi^\sharp_\ell\rangle_{\ell\in\mathrm{Loc}(P)}$ of abstract environments, one for each location ℓ in the program P considered. We provide the set of all abstract surface configurations with a lattice structure by pointwise extension of operations from $E^\sharp$. The denotation $\gamma\langle\nu^\sharp_\ell, \pi^\sharp_\ell\rangle_{\ell\in\mathrm{Loc}(P)}$ of ans abstract configuration is the set of all surface configurations $\langle\lambda, \varrho, \ell\rangle$ such that $\langle\lambda, \varrho\rangle \in \gamma_{E^\sharp}\langle\nu^\sharp_\ell, \pi^\sharp_\ell\rangle$.

Following the methodology of Abstract Interpretation, we must now define the abstract semantics of the language. We first have to define some operations on the abstract numerical lattice $\mathcal{V}^\sharp$. If $\nu^\sharp \in \mathcal{V}^\sharp$ and V is a set of variables, we denote by $\nu^\sharp \ominus V$ the abstract numerical relation in which all information about variables in V has been lost, and by $\lfloor\nu^\sharp\rfloor_V$ the relation that only keeps

information for variables in V. If S is a system of arbitrary numerical constraints, we denote by $\nu^\sharp \ominus S$ an abstract numerical relation representing all variable assignments that are in the denotation of $\nu^\sharp$ and that are also solutions of S. If v is a variable, we denote by $\nu^\sharp[v := v + c]$ the operation that consists of adding the increment c to the value of v. The implementation of these operations depends on the abstract numerical lattice considered, and we refer the reader to the corresponding papers for more details about the underlying algorithms [CC76,Kar76,CH78,Gra91,Min01]. We assign an abstract semantics $[\![s]\!]^\sharp : E^\sharp \to E^\sharp$ to each atomic statement s of the language as defined in Fig. 4.

If $\langle \nu^\sharp, \pi^\sharp \rangle$ is an abstract environment, we define the result $\langle \bar{\nu}^\sharp, \bar{\pi}^\sharp \rangle$ of the operation $\mathbf{inc}_\Lambda \langle \nu^\sharp, \pi^\sharp \rangle$ as follows:

- $\bar{\nu}^\sharp = \nu^\sharp[\Lambda := \Lambda + 1]$
- $\forall \mathrm{p} : \bar{\pi}^\sharp(\mathrm{p}_a) = \begin{cases} \&\mathrm{x} & \text{if } \pi^\sharp(\mathrm{p}_a) = \&\mathrm{x} \\ \mathbf{blk}^\sharp_\ell \langle \mu^\sharp[\Lambda := \Lambda + 1] \rangle & \text{if } \pi^\sharp(\mathrm{p}_a) = \mathbf{blk}^\sharp_\ell \langle \mu^\sharp \rangle \\ \mathbf{ref}^\sharp_\ell \langle \mu^\sharp[\Lambda := \Lambda + 1] \rangle & \text{if } \pi^\sharp(\mathrm{p}_a) = \mathbf{ref}^\sharp_\ell \langle \mu^\sharp \rangle \end{cases}$

We define the operation $\mathbf{dec}_\Lambda \langle \nu^\sharp, \pi^\sharp \rangle$ (resp. $\mathbf{reset}_\Lambda \langle \nu^\sharp, \pi^\sharp \rangle$) similarly by substituting the operation $\Lambda := \Lambda - 1$ (resp. $\Lambda := 0$) to $\Lambda := \Lambda + 1$. The abstract semantics of a program is then given by the least solution of a recursive system of semantic equations

$$\langle \nu^\sharp_\ell, \pi^\sharp_\ell \rangle = F_\ell \left(\langle \nu^\sharp_{\ell'}, \pi^\sharp_{\ell'} \rangle_{\ell' \in \mathrm{Loc}(P)} \right)$$

where F_ℓ is defined as follows:

- If $\ell = \mathbf{next}(\ell')$ and ℓ' is the location of an atomic statement s, then

$$F_\ell \left(\langle \nu^\sharp_{\ell'}, \pi^\sharp_{\ell'} \rangle_{\ell' \in \mathrm{Loc}(P)} \right) = [\![s]\!]^\sharp \langle \nu^\sharp_{\ell'}, \pi^\sharp_{\ell'} \rangle$$

- If $\ell'' : \mathtt{while}\ (\mathtt{m} < \mathtt{n})\ \mathtt{do}\ \ell : s; \cdots; \ell' : \mathtt{end}$, then

$$F_\ell \left(\langle \nu^\sharp_{\ell'}, \pi^\sharp_{\ell'} \rangle_{\ell' \in \mathrm{Loc}(P)} \right) = \langle \nu^\sharp_{\ell''} \ominus \{\mathtt{m} < \mathtt{n}\}, \pi^\sharp_{\ell''} \rangle \sqcup \mathbf{inc}_\Lambda \langle \nu^\sharp_{\ell'}, \pi^\sharp_{\ell'} \rangle$$

- If $\ell = \mathbf{next}(\ell')$ and $\ell' : \mathtt{while}\ (\mathtt{m} < \mathtt{n})\ \mathtt{do}\ \ldots\ \mathtt{end}$, then

$$F_\ell \left(\langle \nu^\sharp_{\ell'}, \pi^\sharp_{\ell'} \rangle_{\ell' \in \mathrm{Loc}(P)} \right) = \begin{cases} \mathbf{reset}_\Lambda \langle \nu^\sharp_{\ell'} \ominus \{\mathtt{m} \geq \mathtt{n}\}, \pi^\sharp_{\ell'} \rangle & \text{if } \mathbf{top}(\ell') \\ \mathbf{dec}_\Lambda \langle \nu^\sharp_{\ell'} \ominus \{\mathtt{m} \geq \mathtt{n}\}, \pi^\sharp_{\ell'} \rangle & \text{otherwise} \end{cases}$$

We apply classical fixpoint algorithms based upon iteration sequences with widening and narrowing [Cou81,CC92] in order to obtain an upper approximation $\mathcal{S}^\sharp$ of the least fixpoint of the system.

Theorem 1. *$\mathcal{S}^\sharp$ is a sound approximation of the surface semantics, i.e. $\mathcal{S} \subseteq \gamma \left(\langle \nu^\sharp_\ell, \pi^\sharp_\ell \rangle_{\ell \in \mathrm{Loc}(P)} \right)$.*

For example, consider the following program in our core pointer language that fills in an array a of pointers with newly allocated blocks:

Example 3.

```
1:  n = 0;
2:  while (n < 10) {
3:    q = &a;
4:    p = q + n;
5:    r = malloc();
6:    *p = r;
7:    n = n + 1;
8:  }
```

If we use the lattice of convex polyhedra [CH78] as the numerical lattice $\mathcal{V}^\sharp$, then the abstract environment obtained after analysis of the surface structure at program point 6 is:

$$\left\langle \begin{cases} 0 \leq \mathtt{n} < 10 \\ \Lambda = \mathtt{n} \\ \mathtt{q}_o = \mathtt{r}_o = 0 \\ \mathtt{p}_o = 4 \times \mathtt{n} \end{cases}, \begin{cases} \mathtt{p}_a \mapsto \{\&\mathtt{a}\} \\ \mathtt{q}_a \mapsto \{\&\mathtt{a}\} \\ \mathtt{r}_a \mapsto \{\mathbf{blk}_S^\sharp\langle \tau = \Lambda, 0 \leq \Lambda < 10\rangle\} \end{cases} \right\rangle$$

assuming that pointers occupy four bytes in memory.

4 Nonuniform Inclusion Constraints

We now use the analysis of the surface structure to build a global approximation of the memory graph. For this purpose we use an extension of Andersen's inclusion constraints [And94] enriched with numerical indices that allow us to describe nonuniform points-to relations. The syntax of a nonuniform inclusion constraint is the following:

$$\begin{aligned} Cst ::= & \langle \mathcal{X}(t) \supseteq \&\mathtt{x} + o, \nu^\sharp(t, o)\rangle \\ \mid & \langle \mathcal{X}(t) \supseteq \mathbf{blk}_\ell\langle t'\rangle + o, \nu^\sharp(t, t', o)\rangle \\ \mid & \langle \mathcal{X}(t) \supseteq \mathcal{Y}(t') + o, \nu^\sharp(t, t', o)\rangle\rangle \\ \mid & \langle *\mathcal{X}(t) \supseteq \mathcal{Y}(t'), \nu^\sharp(t, t')\rangle \\ \mid & \langle \mathcal{X}(t) \supseteq *\mathcal{Y}(t'), \nu^\sharp(t, t')\rangle \end{aligned}$$

where t, t', o are special index variables denoting timestamp and offset values and $\mathcal{X}, \mathcal{Y}$ are set variables. We assume that we are provided with a countable collection of set variables. The second component $\nu^\sharp$ of a nonuniform constraint is a system of numerical relationships between the index variables appearing in the constraint.

The semantics of a system of nouniform constraints is based upon an abstract memory graph. An abstract memory graph $M^\sharp$ is a set of abstract points-to relations

$$\langle \mathtt{a}(t, o) \rhd \mathtt{a}'(t', o'), \nu^\sharp(t, t', o, o')\rangle$$

where $\mathtt{a}, \mathtt{a}'$ are addresses and t, t', o, o' are special index variables representing the timestamps and offsets associated to each address. The abstract numerical

relation $\nu^\sharp$ expresses numerical constraints between these index variables. The set $\mathcal{M}^\sharp$ of abstract memory graphs can be provided with the structure of a lattice by pointwise extension of the corresponding lattice operations over $\mathcal{V}^\sharp$. The denotation $\gamma_{\mathcal{M}^\sharp}(M^\sharp)$ of an abstract memory graph is the set of memory graphs such that the offsets on the points-to edges satisfy the constraints of the corresponding abstract edges. A valuation $V^\sharp$ of set variables is a set of mappings

$$\langle \mathcal{X}(t) \mapsto \mathsf{a}(t') + o, \nu^\sharp(t, t', o) \rangle$$

where a is an address and t, t', o are numerical index variables. The set $Val^\sharp$ of all valuations can similarly be provided with the structure of a lattice. Note that in the case of the address of a global &x, the associated timestamp variable does not have any meaning and is not related by any numerical constraint. We use a uniform notation in order to keep the semantic definitions simple. A valuation can be seen as an abstraction of the anchorage structure defined in Sect. 2. The semantics $[\![C]\!]^\sharp : \mathcal{M}^\sharp \times Val^\sharp \to M^\sharp \times Val^\sharp$ of a nonuniform inclusion constraint C is defined as follows:

- $[\![\langle \mathcal{X}(t) \supseteq \&\mathrm{x} + o, \nu^\sharp \rangle]\!]^\sharp(M^\sharp, V^\sharp) = (M^\sharp, V^\sharp \sqcup \{\langle \mathcal{X}(t) \mapsto \&\mathrm{x} + o, \nu^\sharp \rangle\})$
- $[\![\langle \mathcal{X}(t) \supseteq \mathsf{blk}_\ell\langle t' \rangle + o, \nu^\sharp \rangle]\!]^\sharp(M^\sharp, V^\sharp) = (M^\sharp, V^\sharp \sqcup \{\langle \mathcal{X}(t) \mapsto \mathsf{blk}_\ell\langle t' \rangle + o, \nu^\sharp \rangle\})$
- $[\![\langle \mathcal{X}(t) \supseteq \mathcal{Y}(t') + o, \nu^\sharp \rangle\rangle]\!]^\sharp(M^\sharp, V^\sharp) = (M^\sharp, V^\sharp \sqcup \{\langle \mathcal{X}(t) \mapsto \mathsf{a}(t'') + o'', \lfloor \nu^\sharp \sqcap \mu^\sharp \oplus \{o'' = o + o'\} \rfloor_{t,t'',o''} \rangle \mid \langle \mathcal{Y}(t') \mapsto \mathsf{a}(t'') + o', \mu^\sharp \rangle \in V^\sharp\})$
- $[\![\langle *\mathcal{X}(t) \supseteq \mathcal{Y}(t'), \nu^\sharp \rangle]\!]^\sharp(M^\sharp, V^\sharp) = (M^\sharp \sqcup \{\langle \mathsf{a}(t, o) \rhd \mathsf{a}'(t', o'), \nu^\sharp \sqcap \nu_1^\sharp \sqcap \nu_2^\sharp \rangle \mid \langle \mathcal{X}(t) \mapsto \mathsf{a}(t) + o, \nu_1^\sharp \rangle \in V^\sharp \wedge \langle \mathcal{Y}(t) \mapsto \mathsf{a}'(t') + o', \nu_2^\sharp \rangle \in V^\sharp, V^\sharp)$
- $[\![\langle \mathcal{X}(t) \supseteq *\mathcal{Y}(t'), \nu^\sharp \rangle]\!]^\sharp(M^\sharp, V^\sharp) = (M^\sharp, V^\sharp \sqcup \{\langle \mathcal{X}(t) \mapsto \mathsf{a}'(t''') + o', \mu^\sharp \rangle \mid \langle \mathcal{Y}(t') \mapsto \mathsf{a}(t'') + o, \nu_1^\sharp \rangle \in V^\sharp \ \wedge \ \langle \mathsf{a}(t'', o) \rhd \mathsf{a}'(t''', o'), \nu_2^\sharp \rangle \in M^\sharp \ \wedge \ \mu^\sharp = \lfloor \nu^\sharp \sqcap \nu_1^\sharp \sqcap \nu_2^\sharp \rfloor_{t,t''',o'}\})$

where we have freely renamed the index variables whenever it was necessary to avoid name clashes. A solution of a system S of nonuniform set constraints is a couple $(M^\sharp, V^\sharp)$ which is invariant under the application of $[\![C]\!]^\sharp$ for any $C \in S$.

We are interested in the least solution of a system S of nonuniform set constraints. We can obtain an approximation of the least solution of S by computing the limit of the abstract iteration sequence with widening $(M_n^\sharp, V_n^\sharp)_{n \geq 0}$ defined as follows:

$$\begin{cases} (M_0^\sharp, V_0^\sharp) & = (\bot_{\mathcal{M}^\sharp}, \bot_{Val^\sharp}) \\ (M_{n+1}^\sharp, V_{n+1}^\sharp) & = (M_n^\sharp, V_n^\sharp) \nabla \left([\![C]\!]^\sharp\right)_{C \in S}^{\sim}(M_n^\sharp, V_n^\sharp) \end{cases}$$

where $\left([\![C]\!]^\sharp\right)_{C \in S}^{\sim}$ denotes the application of all constraints of S in an arbitrary order, and ∇ is the product of the widening operators on $\mathcal{M}^\sharp$ and $Val^\sharp$. This provides us with an effective algorithm for computing an approximate solution of the system, which is similar to that defined by Andersen [And94]. The main difference is the use of a widening operator to enforce convergence because some abstract numerical lattices have infinitely increasing chains of elements[CC76,CH78,Min01]. Once a post-fixpoint has been reached using this

algorithm, we can further refine the result by using a decreasing iteration sequence with narrowing defined in the same way. We observed from our experiments that an iteration sequence with narrowing is always required in order to obtain precise ranges for the timestamp and offset variables.

We now have to show how to extract nonuniform inclusion constraints from the abstract interpretation of the surface semantics. Let $\mathcal{S}^\sharp$ be the abstract surface semantics of a program P obtained from the analysis described in the previous section. We assign a unique pair of set variables $(\mathcal{L}_\ell, \mathcal{R}_\ell)$ to each statement $\ell : *\mathtt{q} = \mathtt{r}$ or $\ell : \mathtt{q} = *\mathtt{r}$ of P, denoting respectively the points-to sets of the lefthand and righthand sides of the assignment. Let $\varrho^\sharp = \langle \nu^\sharp, \pi^\sharp \rangle$ be an abstract environment, $\mathtt{p}$ a pointer variable of P and $\mathcal{X}$ a set variable. We denote by $C_{\mathcal{X},\mathtt{p}}\langle\varrho^\sharp\rangle$ the collection of nonuniform constraints defined as follows:

- If $\&\mathtt{x} \in \pi^\sharp(\mathtt{p}_a)$, then

$$\langle \mathcal{X}(t) \supseteq \&\mathtt{x} + o, \lfloor \nu^\sharp \oplus \{t = \Lambda, o = \mathtt{p}_o\} \rfloor_{t,o} \rangle \in C_{\mathcal{X},\mathtt{p}}\langle\varrho^\sharp\rangle$$

- If $\mathbf{blk}^\sharp_\ell\langle\mu^\sharp\rangle \in \pi^\sharp(\mathtt{p}_a)$, then

$$\langle \mathcal{X}(t) \supseteq \mathbf{blk}_\ell\langle t' \rangle + o, \lfloor \nu^\sharp \sqcap \mu^\sharp \ominus \{\tau = t', t = \Lambda, o = \mathtt{p}_o\} \rfloor_{t,t',o} \rangle \in C_{\mathcal{X},\mathtt{p}}\langle\varrho^\sharp\rangle$$

- If $\mathbf{ref}^\sharp_\ell\langle\mu^\sharp\rangle \in \pi^\sharp(\mathtt{p}_a)$, then

$$\langle \mathcal{X}(t) \supseteq \mathcal{L}_\ell(t') + o, \lfloor \nu^\sharp \sqcap \mu^\sharp \oplus \{\tau = t', t = \Lambda, o = \mathtt{p}_o\} \rfloor_{t,t',o} \rangle \in C_{\mathcal{X},\mathtt{p}}\langle\varrho^\sharp\rangle$$

Now, if $\ell : *\mathtt{p} = \mathtt{q}$ is a memory write statement of P and $\varrho^\sharp$ is the abstract environment of $\mathcal{S}^\sharp$ at ℓ, we generate the constraints:

$$C_{\mathcal{L}_\ell,\mathtt{p}}\langle\varrho^\sharp\rangle \cup C_{\mathcal{R}_\ell,\mathtt{q}}\langle\varrho^\sharp\rangle \cup \{\langle *\mathcal{L}_\ell(t) \supseteq \mathcal{R}_\ell(t'), \top_{\mathcal{V}^\sharp} \oplus \{t = t'\}\rangle\}$$

Similarly, for a memory read statement $\ell : *\mathtt{p} = \mathtt{q}$ we generate the constraints:

$$C_{\mathcal{L}_\ell,\mathtt{p}}\langle\varrho^\sharp\rangle \cup C_{\mathcal{R}_\ell,\mathtt{q}}\langle\varrho^\sharp\rangle \cup \{\langle \mathcal{L}_\ell(t) \supseteq *\mathcal{R}_\ell(t'), \top_{\mathcal{V}^\sharp} \oplus \{t = t'\}\rangle\}$$

We denote by S_P the system of all constraints generated in this way for the program P. Let $(M^\sharp_P, V^\sharp_P)$ be an approximation of the least solution of S_P obtained by an abstract iteration sequence as described previously. The abstract memory graph $M^\sharp_P$ is a sound global approximation of the memory graph at every point of the program:

Theorem 2. *For all state $\langle \lambda, A, M, \varrho, \ell \rangle$ of the collecting semantics C of P, we have $M \in \gamma_{\mathcal{M}^\sharp}(M^\sharp_S)$.*

The pointer analysis problem of [Ven02] has thus been reduced to the simpler and more tractable problem of solving a system of nonuniform inclusion constraints.

We finish this formal description with a brief description of the constraint generation for function calls. We associate a special set variable $\mathcal{F}_i(\mathtt{f})$ to the i-th formal parameter of each function $\mathtt{f}$ of a C program. We denote by $\mathcal{F}_0(\mathtt{f})$ the variable corresponding to the return value of $\mathtt{f}$. Now consider a function

call ℓ : $\mathtt{p} = \mathtt{f}(\mathtt{p}_1, \ldots, \mathtt{p}_n)$. Assuming that we are provided with a collection $\mathcal{X}, \mathcal{X}_1, \ldots, \mathcal{X}_n$ of set variables describing the sets of addresses that may flow through the return value and the parameters $\mathtt{p}, \mathtt{p}_1, \ldots, \mathtt{p}_n$ of the function call, we generate the following points-to equations:

$$\begin{cases} \langle \mathcal{F}_1(\mathtt{f}) \supseteq \mathcal{X}_1, \top_{\mathcal{V}^\sharp} \rangle \\ \ldots \\ \langle \mathcal{F}_n(\mathtt{f}) \supseteq \mathcal{X}_n, \top_{\mathcal{V}^\sharp} \rangle \\ \langle \mathcal{X} \supseteq \mathcal{F}_0(\mathtt{f}), \top_{\mathcal{V}^\sharp} \rangle \end{cases}$$

In other words, function calls are treated *uniformly*: there are no numerical constraints on the index variables. This is not a problem in practice, since nonuniform behaviours usually take place at the function level in embedded applications. We do not detail the analysis of computed calls, which can be easily derived from the semantics of the memory read operation $\mathtt{p} = *\mathtt{q}$.

We now illustrate the generation of equations. Consider the small program of Example 3 that fills in an array of pointers. The equations generated after the surface analysis are the following:

$$\begin{cases} \langle *\mathcal{L}_6(t) \supseteq \mathcal{R}_6(t'), \{t = t', 0 \leq t < 10\} \rangle \\ \langle \mathcal{L}_6(t) \supseteq \&\mathtt{a} + o, \{0 \leq o \leq 4 \times t\} \rangle \\ \langle \mathcal{R}_6(t) \supseteq \mathbf{blk}_5\langle t' \rangle + o, \{t = t', o = 0, 0 \leq t < 10\} \rangle \end{cases}$$

After solving these constraints by using an abstract iteration sequence with widening, we obtain the following abstract memory graph:

$$\{\langle (\&\mathtt{a}, o) \rhd (\mathbf{blk}_5\langle t \rangle, o'), \{o = 4 \times t, o' = 0, 0 \leq t < 10\} \rangle\}$$

which describes the exact shape of the memory althrough the execution of the program.

5 Experimental Evaluation

We have implemented the static analysis described in this paper for the full C language. The analyzer itself consists of 9,000 lines of SML/NJ excluding the front-end. We have interfaced the analyzer with the ckit [HOM] C front-end which is also written in SML. We currently use the reduced product of the lattice of linear equalities [Kar76] and the lattice of intervals [CC76] for expressing numerical constraints. The analyzer first translates the C program into an intermediate language in which all expressions and statements have been broken down using a 3-address format. We then perform a dependency analysis which is used to eliminate all arithmetic operations that are not involved in pointer manipulations. This substantially shrinks down the size of the code to analyze. Whole structure assignment has not been described in this paper and deserves some attention. There are two possible ways of handling this construct, either by expanding the assignment into a collection of individual assignments to the fields of the structure or by analyzing the assignment as an atomic operation.

The former is made difficult by union types and structure-breaking type casts. We chose the latter approach, which requires a straightforward extension of nonuniform constraints in order to copy a packet of pointers at once.

We have applied the analyzer to a real piece of software: an on-board link controller. The application contains about 25,000 lines of unprocessed C code. It is a pointer intensive program with plenty of loop constructs operating on multidimensional arrays of structures. It is quite representative of an average size embedded program, which is the main target of our analysis. Very large programs like those described in [VB04] are quite unusual. Our analysis is quite efficient. It takes 210 seconds to parse the files, construct the abstract surface semantics and generate the nonuniform inclusion constraints on a laptop with a 900Mhz Intel Pentium and 1Gb of RAM running Linux under VmWare. The resolution of these constraints only takes 21 seconds.

The results show that the analysis does discover nonuniform points-to relations. In particular, bidimensional arrays of distinct semaphores, arrays of functions and tables of preallocated memory blocks for dedicated memory management are exactly described. Surprisingly enough, the analysis uncovered a real bug in this application. While we were reviewing the results of the analysis we noticed that for some array array2 of dynamically allocated semaphores, there was no linear relationship between the offset and the timestamps in the points-to relations. The nonuniform points-to equations gave us instantly the location in the program where the array was initialized. The initialization code looks like:

```
for (i = 0; i < 20; i++)
  for (j = 0; j < 8; j++) {
    array1[i][j] = semCreate ();
    array2[j] = semCreate ();
  }
```

The first array is properly initialized whereas the second one is reinitialized multiple times, causing a memory leak. It should be noticed that the analysis sucessfully inferred a nonuniform points-to relation for the bidimensional array of semaphores. This bug was present from the very first version of the program and has never been detected during the 18 months the software has been undergoing testing so far. This is an interesting application of this static analysis as a sophisticated typechecker for collections of pointers.

6 Conclusion

We have presented a pointer analysis that is able to infer nonuniform points-to relationships without the cost of existing flow-sensitive analyses [Deu94,Ven02]. The originality of our work is that it conciliates two approaches to pointer analysis, abstract interpretation and constraint-based analysis, which are often opposed one to each other. Although we could have expressed the whole analysis within the framework of Abstract Interpretation [CC95], we think that a

constraint-based presentation is more compact and more intuitive for both understanding and implementing the analysis. We have shown on a representative case study that our approach is tractable and achieves the expected level of precision. Unexpectedly, this analysis has been able to detect a subtle initialization bug in a real application. It now remains to perform more extensive empirical studies and investigate the use of the analysis in a real verification tool.

References

[And94] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[BCC+02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer-Verlag, October 2002.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM Press, June 7–14 2003.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of 2nd International Symposium on Programming*, pages 106–130, 1976.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 238–353, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, New York, NY, 1979.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181. ACM Press, New York, NY, 1995.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, New York, NY, 1978.

[Cou81] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, 1981.

[CR99] Satish Chandra and Thomas W. Reps. Physical type checking for c. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 66–75, 1999.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. *ACM SIGPLAN Notices*, 35(5):35–46, 2000.

[Deu92] A. Deutsch. A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 2–13. IEEE Computer Society Press, 1992.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*. ACM Press, 1994.

[FFSA98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. *ACM SIGPLAN Notices*, 33(5):85–96, 1998.

[Gra91] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT'91*, volume 493. Lecture Notes in Computer Science, 1991.

[HOM] Nevin Heintze, Dino Oliva, and Dave MacQueen. The ckit front-end. `ckit@research.bell-labs.com`.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.

[Min01] A. Miné. The octagon abstract domain. In *AST 2001 at WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[Ste96] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Computational Complexity*, pages 136–150, 1996.

[VB04] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the International Conference on Programming Language Design and Implementation, PLDI'04*, 2004. To appear.

[Ven96] A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Proceedings of SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 266–382. Springer Verlag, 1996.

[Ven99] A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 35(2):223–248, 1999.

[Ven02] A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In *Proceedings of the 9th International Symposium on Static Analysis SAS'02*, volume 2477 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2002.

[WL02] John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium*, pages 180–195, September 2002.